Finding Backdoors to Integer Programs: A Monte Carlo Tree Search Framework

Elias B. Khalil[1, 2], **Pashootan Vaezipoor**[3, 4], **Bistra Dilkina**[5]

[1]Department of Mechanical & Industrial Engineering, University of Toronto
[2]Scale AI Research Chair in Data-Driven Algorithms for Modern Supply Chains
[3]Department of Computer Science, University of Toronto
[4]Vector Institute for Artificial Intelligence
[5]Department of Computer Science, University of Southern California
khalil@mie.utoronto.ca, pashootan@cs.toronto.edu, dilkina@usc.edu

Abstract

In Mixed Integer Linear Programming (MIP), a (strong) *backdoor* is a "small" subset of an instance's integer variables with the following property: in a branch-and-bound procedure, the instance can be solved to global optimality by branching *only* on the variables in the backdoor. Constructing datasets of pre-computed backdoors for widely used MIP benchmark sets or particular problem families can enable new questions around novel structural properties of a MIP, or explain why a problem that is hard in theory can be solved efficiently in practice. Existing algorithms for finding backdoors rely on sampling candidate variable subsets in various ways, an approach which has demonstrated the existence of backdoors for some instances from MIPLIB2003 and MIPLIB2010. However, these algorithms fall short of consistently succeeding at the task due to an imbalance between exploration and exploitation. We propose `BaMCTS`, a Monte Carlo Tree Search framework for finding backdoors to MIPs. Extensive algorithmic engineering, hybridization with traditional MIP concepts, and close integration with the CPLEX solver have enabled our method to outperform baselines on MIPLIB2017 instances, finding backdoors more frequently and more efficiently.

1 Introduction

Hard discrete optimization problems arise in a very wide range of application domains. While the theoretical computer science approach focuses on the design of algorithms with approximation guarantees, many real problems are not amenable to such analyses. In the artificial intelligence and operations research communities, on the other hand, much of the focus is on deriving algorithms that can efficiently produce high-quality solutions (even if without approximation bounds) and/or optimality guarantees (even when the worst-case running time is exponential). In this work, we focus on the latter class of approaches, in particular the branch-and-bound procedure for the exact solution of Mixed Integer Programming (MIP) problems. MIP solvers are celebrated for their ability to solve problems with hundreds of thousands to millions of variables and constraints, a feat which the theory would suggest to be impossible in a reasonable amount of time. Yet, little is known about *why* realistic instances of NP-Hard MIPs can be solved relatively efficiently in practice.

The existence of *backdoor* sets provides one possible answer to this puzzle of empirical tractability in exact MIP solving. This was first reported by Dilkina et al. (2009) who showed that backdoors of sizes 10–20 did exist for some instances (of MIPLIB2003 (Achterberg, Koch, and Martin 2006)) with many hundreds to thousands of integer variables. The intuition is simple: if one need only branch on a handful of integer variables to solve a MIP instance to global optimality (or declare it as infeasible), then it is not difficult to imagine that a full-fledged MIP solver would be able to solve that instance without much branching. This motivates the design of algorithms for finding backdoors, with the following potential use cases: (i) Discovering heretofore unknown structural properties for a family of MIP instances, e.g., a new variant of an independent set or facility location problem, by data-mining backdoors from a large number of instances; (ii) Constructing datasets of pre-computed backdoors that can then be used to train machine learning models for quickly identifying these crucial sets on similar but unseen instances and branching on them for a quick solving time; (iii) Directly using the backdoor for branching, assuming that the backdoor in question can be computed quickly in advance; such a use case, if realized, could speed up solving times substantially for instances that are challenging for current branching strategies in state-of-the-art MIP solvers.

How, then, does one find backdoors? We will denote the desired backdoor size by $K \in \mathbb{N}^+$. Despite negative computational complexity results (e.g., Szeider (2005) shows that finding backdoors for SAT requires time exponential in K, even for fixed K), some heuristics have been developed. Dilkina et al. (2009) used two forms of *random sampling*: independent *uniform* sampling of K integer variables, and independent *biased* sampling of K integer variables to favor ones that are more fractional (i.e., with fractional parts closer to 0.5) in the solution of the Linear Programming (LP) relaxation of the MIP. Candidate sets are sampled (in parallel) and used for branching in a MIP solver; if global optimality is attained, then a backdoor has been found. The biased sampling strategy proved much more effective than uniform. While useful as a proof-of-concept, random sampling is a *pure exploration* strategy which cannot use knowledge from previous draws to focus future ones towards more promising candidate sets.

Fischetti and Monaci (2011) propose another strategy that leverages the following basic fact about (bounded) mixed 0–1 problems: the set of extreme points (vertices) of the LP-feasible region includes all integer-feasible points. The branch-and-bound tree that results from branching on a strong backdoor must have leaf nodes whose LPs are either infeasible or have integer-feasible optima. This leads to the following equivalent definition for a strong backdoor: a small set $\mathcal{B}$ of 0–1 variables such that each fractional vertex has at least one of its fractional variables in $\mathcal{B}$. If one could enumerate all (exponentially many) fractional vertices, then a Set Covering Problem (SCP) can be formulated to find a backdoor. In practice, the fractional vertices are collected progressively in batches by executing branch-and-bound using the current candidate backdoor. An SCP is solved at each iteration until all fractional vertices have been "covered" or an early termination criterion is met. This method can be extended heuristically for general MIPs. Fischetti and Monaci (2011) demonstrate that this method's candidate backdoors, which cover many but rarely all fractional vertices, can lead to smaller trees if selected as branching variables early on in the search. However, because (i) there can be many fractional vertices and (ii) the order in which they are explored is arbitrary (see (Fischetti and Monaci 2014; Fischetti 2014) for a discussion), it is unclear if the information that this method uses is actually conducive to backdoors in practice, i.e., it may be *"exploiting" too aggressively*. Additionally, if one is interested in backdoors of size at most K (as is the case in this paper), this approach may be unable to produce a solution as the SCP requires covering all of the collected fractional vertices.

Contributions Thus far, we have argued (i) that finding backdoors is beneficial for a variety of tasks and (ii) that existing algorithms either explore or exploit too much. Towards a more general, effective, and extensible backdoor search algorithm, we propose `BaMCTS` (short for "Backdoor MCTS"), a Monte Carlo Tree Search (MCTS) framework for finding backdoors to MIPs. Our contributions can be summarized as follows:

1. **Backdoor Search as MCTS:** We contribute the first such formulation of the problem. `BaMCTS` can balance exploration and exploitation by design, is conceptually simple and easy to implement, and is extensible in a plug-and-play fashion.
2. **Tight Integration with MIP Domain Knowledge:** To enable a scalable and effective solution, we customize the high-level MCTS procedure to the MIP setting through the use of domain-specific reward functions, action scoring functions and elimination rules, and careful engineering that is tightly coupled with CPLEX, a widely used MIP solver.
3. **Extensive Empirical Evaluation:** reveals that `BaMCTS` vastly outperforms the sampling strategy of Dilkina et al. (2009), finding "better" backdoors in a shorter amount of time on instances from the MIPLIB2017 Benchmark set (Gleixner et al. 2021). We also show that branching on such sets of variables results in smaller search trees or optimality gaps compared to CPLEX with its default branching.

2 Related Work

Backdoors for Combinatorial Problems The notion of backdoors was first introduced by Williams, Gomes, and Selman (2003) for SAT, where it was observed that practical SAT instances often have a small tractable structures. Over the years many approaches were proposed to find backdoors in the SAT context (Paris et al. 2006; Kottler, Kaufmann, and Sinz 2008; Li and Van Beek 2011). Observing the connection between SAT and MIP, Dilkina et al. (2009) generalized the concept of backdoors from SAT to MIP and proposed random sampling as a method for finding backdoors. Fischetti and Monaci (2011) proposed another strategy for MIP, which to our knowledge remains the state-of-the-art to this day in the MIP context.

MCTS and Applications MCTS has seen a surge of interest thanks to its great success particularly in solving two-player games (Silver et al. 2017). This has led to many attempts in solving combinatorial optimization problems using MCTS by translating the problem into a game. In particular, UCT has been used to guide MIP solvers (Sabharwal, Samulowitz, and Reddy 2012) and to solve Quantified Constraint Satisfaction Problems (QCSP) (Satomi et al. 2011). Bertsimas et al. (2014) applied MCTS to the challenging problem of Dynamic Resource Allocation (DRA) and showed that it can greatly improve problem-specific baselines on large scale instances. The Travelling Salesman Problem (TSP), another classical problem, was addressed in Rimmel, Teytaud, and Cazenave (2011) via a version of Monte Carlo search with some success. Bandit Search for Constraint Programming (BaSCoP) (Loth et al. 2013) applied MCTS to improve the tree-search heuristics of Constraint Programmin (CP) solvers and showed significant improvements on the depth-first search on certain CP benchmarks. More recent works involve successful application of the "neural" MCTS of AlphaZero to solve a variety of NP-Hard graph problems (Abe et al. 2019), as well as solving First-Order Logic descriptions of combinatorial problems (Xu, Kadam, and Lieberherr 2021).

The difference between our setting and a typical combinatorial optimization problem is that our reward function is an expensive black box, namely one that requires running a MIP solver for a limited number of steps, as opposed to an analytical objective function; this makes our setting more challenging because evaluations are time-consuming.

3 Technical Background

3.1 Branch-and-Bound for MIP

We are concerned with Mixed Integer Linear Programming (MIP) problems of the form:

$$z^* = \min\{c^T x | Ax \leqslant b, x \in \mathbb{R}^n, x_j \in \mathbb{Z}\ \forall j \in I\},$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and the non-empty set $I \subseteq \{1, ..., n\}$ indexes the integer variables. The vectors in the set $X_{MIP} = \{x \in \mathbb{R}^n | Ax \leqslant b, x_j \in \mathbb{Z}\ \forall j \in I\}$

are *integer-feasible solutions*. An integer-feasible (or simply feasible) solution $x^* \in X_{MIP}$ is *optimal* if $c^T x^* = z^*$.

A MIP can be solved by *branch and bound* (Cook 2012), an exact algorithm that divides the original MIP into sub-problems organized in a binary tree; see Nemhauser and Wolsey (1988) for a textbook exposition. At each node of the tree, an LP relaxation of the sub-problem is solved. If the resulting solution x_N of the LP relaxation at a node N is integral, then it is also a feasible solution to the MIP, i.e., $x_N \in X_{MIP}$. If such an integral solution has an objective value that is better than the best one found so far, it is referred to as the *incumbent*, maintaining that designation until a better solution is found. Otherwise, the node is either pruned, if its lower bound is greater than the incumbent's value, or branched on, resulting in two child nodes that are added to the queue of nodes to be processed.

Pseudocosts are historical quantities, aggregated for each variable during the search, that represent the amount by which a node's LP relaxation value has been tightened when branching on a given variable. A higher pseudocost score indicates that branching on a variable typically helps make progress towards proving optimality. As such, pseudocosts are at the heart of most typical branching strategies that are used in MIP solvers (Achterberg, Koch, and Martin 2005; Achterberg and Berthold 2009; Hendel 2015).

3.2 Backdoors for MIP

Given a MIP instance defined by the tuple (A, b, c, I), a *strong backdoor* of size $K \lll |I|$ is a set of integer variables $B, |B| = K, B \subset I$ such that branching exclusively on variables from B results in a provably optimal solution or a proof of infeasibility. We will consider *order-sensitive* strong backdoors, where B is an ordered set: a variable at rank i must be branched on before variables of rank $j > i$. The order in which the backdoor variables are considered for branching can affect the performance of the solver's primal heuristics, which in turn affects pruning in branch-and-bound. We refer to Dilkina et al. (2009) for a detailed discussion of why order matters in practice in MIP solvers. Throughout the paper, we will use the term "candidate backdoor" to refer to an ordered set of integer variables that is being assessed but that may or may not be a strong backdoor.

3.3 Monte Carlo Tree Search (MCTS)

MCTS is a randomized algorithm for sequential games with a finite horizon and a finite number of actions n (Browne et al. 2012). At every step of a two-player game, MCTS seeks to identify the next action to take so as to maximize the probability of winning the game. It does so by building out an n-ary tree in which the root node represents the current state of the game, each edge represents a valid action, and a child node represents the extension of its parent's state by playing the action of the corresponding edge. Associated with each terminal state is a scalar reward value.

Because the complete search tree is exponential in size, MCTS is organized into four key steps that can together navigate the exploration-exploitation trade-off in a sensible way, building out only a partial search tree that is sufficient for identifying a good next action. The four steps are:

(a) **Selection:** Given the current search tree, this step deals with *selecting* the nodes in a depth-first dive from the root. Upper Confidence Trees (UCT) (Kocsis and Szepesvári 2006) is a widely used scoring rule that combines the average observed reward of a node (or state) with a function of the number of visits to the node; nodes with large average rewards and/or small visit counts obtain high UCT scores, balancing exploration and exploitation. The latter is controlled by an appropriately tuned hyperparameter.

(b) **Expansion:** When the selected node has only a subset of its potential children as nodes in the current search tree, one can choose to *expand* the selected node's child set by creating a node for a new action. When the number of possible actions is large, Progressive Widening (Coulom 2007; Couëtoux et al. 2011) is used to limit the branching factor of the tree and focus on expanding only frequently-visited (likely more promising) nodes. To select a new action to expand with, uniform sampling may be used. A more "exploitative" expansion would select an action that has led to high-reward children in other nodes of the search tree.

(c) **Simulation:** Following the depth-first dive from the root through a sequence of selections (and possibly a final expansion), a non-terminal node may be reached. Because rewards are only observed in terminal states, e.g., when the game concludes and a winner is declared, *simulation* is used to traverse the state space from the node in question to a terminal state, without consolidating this sub-path into the search tree. Typically, one picks actions randomly until a terminal state is reached.

(d) **Backpropagation:** Following the simulation, a reward is collected. *Backpropagation* refers to the credit assignment process whereby the reward is passed on to the nodes along the depth-first path in the tree that led to the observed reward. A sum-backup rule is commonplace: each node accumulates rewards every time it is selected. Alternatively, a more aggressive max-backup rule keeps track of the maximum observed reward, see (Sabharwal, Samulowitz, and Reddy 2012) for example.

The four-step loop is repeated until a termination condition (e.g., time limit) is reached, following which the action corresponding to the highest-reward or most visited child of the root node is "played". The opponent responds, and the process is repeated starting with a new search tree representing the updated state of the game.

4 `BaMCTS`[1]

Rather than treat MCTS as a black-box algorithm, we instantiate it for the backdoor search setting by incorporating as much domain knowledge about MIP solving as possible.

We are given a MIP instance defined by the tuple (A, b, c, I) and assume a user-defined bound on the backdoor size, $K \in \mathbb{N}^+$; $K \lll |I|$ should be seen as a small con-

[1]Our implementation of `BaMCTS` can be found at: https://github.com/lyeskhalil/backdoorsearch

stant on the order of 5 to 10, which would imply a branch-and-bound tree with hundreds of nodes.

We view backdoor search as a *single-player*, *deterministic* constraint satisfaction game. The goal is to find an order-sensitive strong backdoor that satisfies the size-K constraint on the backdoor size.

We now define two key elements of MCTS for the backdoor setting. Let $P(I, K)$ denote the set of all permutations of all subset of the integer set I that have size at most K. For example, if $K = 2$ and $I = \{1, 2, 3\}$ then $P(I, K) = \{(), (1), (2), (3), (1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)\}$.

- **State:** A state $S \in P(I, K)$ is a permutation of the variables s.t. $|S| \leq K$. A state is terminal iff $|S| = K$. The root of the MCTS tree corresponds to the empty state $()$.
- **Action:** Given a non-terminal state S, a valid action is the index $i \in I$ of an integer variable such that $i \notin S$. Taking action i in state S means that i is appended to the end of the ordered set S, leading to a new state $S' \in P(I, K), |S'| = |S| + 1$ in which the first $|S|$ variables are the same as those in S.

4.1 Building Blocks

Candidate Evaluation Consider a terminal state $\hat{S}$. To evaluate this candidate, a MIP solver is used to check if it is a backdoor. In particular, we instrument the CPLEX solver to (i) branch only on variables in $\hat{S}$, terminating the branch-and-bound if either the instance is solved (i.e., a backdoor has been found) or branching is no longer possible for at least one subproblem (a leaf node on the frontier of the branch-and-bound tree); (ii) respect the ordering implied by $\hat{S}$; (iii) collect auxiliary data such as pseudocosts and search completion information that will be useful for action selection and reward computation.

Reward Shaping The goal – finding an order-sensitive strong backdoor – suggests a binary reward function in which a value of 1 is assigned to a terminal state during MCTS iff a backdoor is found, after which the search terminates because the goal has been achieved. However, this sparse reward structure makes any form of focused search impossible, an issue that we will tackle by using an appropriate reward function which assigns an informative score to a terminal state that is not a backdoor.

A good reward function is one that gives a large, but not maximal, value to a candidate which satisfies most of the conditions that define a strong backdoor. What are those "conditions"? Following backdoor candidate evaluation using the MIP solver, a (potentially incomplete) branch-and-bound tree (not to be confused with the MCTS tree) can be observed. For the candidate to be a backdoor, all leaf nodes of the tree must be closed, i.e., they must have LP relaxations that are either infeasible, integer-feasible, or fathomed by bound; these are the conditions that must be simultaneously satisfied. Now consider a candidate which satisfies most but not all leaf node conditions. The *tree weight* (Kilby et al. 2006) is a scoring function that maps a branch-and-bound tree to a value in $[0, 1]$. When all integer variables are binary, the tree weight is defined as the fraction of binary assignments that belong to a closed leaf node; a strong backdoor would have a tree weight of 1 because all leaf nodes are closed and thus all binary assignments are covered. Importantly, the tree weight achieves our desired criterion for a good reward function: it can give meaningful scores to candidate backdoors that are not true strong backdoors but that eliminate many binary assignments.

Consider the (binary) search tree of branch-and-bound for a mixed-binary integer program, and let T_k denote the tree at the k-th iteration, i.e., after k nodes have been expanded. Let F_k denote the subset of tree nodes that are "final" or fathomed due to their LP relaxations being infeasible, mixed-binary feasible, or worse in value than the best integer-feasible solution's value at the time they were expanded. The tree weight assigns to each node $v \in F_k$ a *weight* of $2^{-d(v)}$, where $d(v)$ is the depth of v. The total tree weight of tree T_k then writes:

$$\text{tree-weight}(T_k) = \sum_{v \in F_k} 2^{-d(v)}.$$

At the start of branch-and-bound, no nodes have been fathomed and so $F_0 = \emptyset, \text{tree-weight}(T_0) = 0$. This function strictly increases with more fathomed nodes. We refer to section 4.3 of (Hendel et al. 2021) for further details and an illustrative example, and note that other tree search completion metrics therein could potentially be used instead of tree weight.

The Role of Pseudocosts Pseudocosts are historical quantities aggregated for each variable during the search. The upwards (downwards) PC of a variable x_j is the average unit objective gain taken over upwards (downwards) branchings on x_j in previous nodes; we refer to this quantity as Ψ_j^+ (Ψ_j^-). Pseudocost branching at node N with LP solution $\check{x}$ consists in computing values:

$$PC_j = \text{score}\Big((\check{x}_j - \lfloor \check{x}_j \rfloor)\Psi_j^-, (\lceil \check{x}_j \rceil - \check{x}_j)\Psi_j^+\Big)$$

and choosing the variable with the largest such value. Typically, the product is used to combine the downwards and upwards values. One standard way to initialize the pseudocost values is by applying strong branching once for each integer variable, at the first node at which it is fractional (Linderoth and Savelsbergh 1999). We will refer to this PC strategy with strong branching initialization as *pseudocost branching* (PC).

As mentioned earlier, pseudocosts play a big role in most MIP branching strategies. Because the definition of a backdoor relies on branching, it is natural to consider ways in which pseudocosts can help steer `BaMCTS` towards promising variables. Indeed, we will show a bit later how pseudocosts can serve as *global scores* for actions (variables). Those global scores are then naturally incorporated into the selection and expansion steps of MCTS. For now, we emphasize that `BaMCTS` tracks the pseudocosts resulting from each backdoor candidate evaluation. We then maintain, for each variable, a running average of its pseudocost score across all candidate evaluations that involved the variable.

Action Space Reduction We would like BaMCTS to scale to MIPs with tens or hundreds of thousands of integer variables. However, the MCTS tree grows fast with the number of integer variables, which may hamper progress. To reduce the action space, we leverage the empirical observation that only a few of the integer variables take on fractional values in the solution of the LP relaxation of the MIP; for instance, Berthold (2014) shows that, on average across 159 instances from older MIPLIB instance libraries, 71.7% of the integer variables are integer in the LP relaxation solution. Rather than work with the full integer set I, we restrict the action space to the subset $I_{\text{frac}} \subseteq I$ of variables that are fractional in the LP relaxation of the MIP instance. While heuristic, this restriction has some grounds in empirical MIP solving and reduces the action space dramatically.

4.2 Instantiating the Four Steps

Selection We adopt a variant of the UCT selection rule, inspired by Gaudel and Sebag (2010), that adds a global action score (average pseudocosts in our case) to UCT's typical elements. Consider an MCTS search tree node (or state) S whose child node S' is being assessed; assume S' extends S with variable $i \in I_{\text{frac}}$.

We let T_S denote the number of visits to node S; $\text{EXP}(S, S')$ denotes the exploration score of S', which is large when $T_{S'}$ is much smaller than T_S; $\hat{\mu}_{S'}$ denotes the current average reward of state S'; $\mathbf{r}_{S'}$ is the vector of rewards that have been observed in the subtree rooted at S' and $\sigma^2(\mathbf{r}_{S'})$ is its variance. Our final scoring function for node selection is given by (1):

$$\text{SCORE}(S, S') = (1-\alpha_{\text{PC}})\text{UCT}_{\text{score}}(S, S') + \alpha_{\text{PC}}\hat{\text{PC}}_i. \quad (1)$$

With $\alpha_{\text{PC}} \in [0, 1)$, the scoring function is a convex combination of a UCT-type score for state S' and the average pseudocost score $\hat{\text{PC}}_i$ of variable i. The latter may be interpreted as a RAVE score following (Gelly and Silver 2007). To arrive at the final scoring function, we define the exploration score (based on the standard UCT formula), the variance score (based on the UCB1-Tuned of Auer, Cesa-Bianchi, and Fischer (2002)), UCT without variance, UCT with variance, and the UCT score which is one of the two preceding scores depending on the value of use_variance, respectively:

$$\begin{aligned}
\text{EXP}(S, S') &= \sqrt{\ln(T_S)/T_{S'}} \\
\text{VAR}(S') &= \sqrt{\min\left\{\frac{1}{4}, \sigma^2(\mathbf{r}_{S'}) + \text{EXP}(S, S')\right\}} \\
\text{UCT}(S, S') &= \hat{\mu}_{S'} + C \cdot \text{EXP}(S, S') \\
\text{UCT}_{\text{var}}(S, S') &= \hat{\mu}_{S'} + C \cdot \text{EXP}(S, S') \cdot \text{VAR}(S') \\
\text{UCT}_{\text{score}}(S, S') &= \begin{cases} \text{UCT}_{\text{var}}(S, S') & \text{if } \texttt{use\_variance}, \\ \text{UCT}(S, S') & \text{otherwise.} \end{cases}
\end{aligned}$$

To conclude, we note that the scoring function has three hyperparameters whose effects will be analyzed experimentally in the next section:

- $\alpha_{\text{PC}} \in [0, 1)$: the weight accorded to the global pseudocost average;
- $C \in \mathbb{R}_{>0}$: the exploration weight;
- use_variance $\in \{\text{True}, \text{False}\}$: a boolean that determines whether UCT with or without variance is used.

Expansion Besides the traditional *uniform random* expansion rule, we consider a deterministic *best score* expansion rule which simply expands using the available action (variable) with the largest average pseudocost score. Together, these two rules cover a wide range along the exploration-exploitation spectrum.

Simulation We opt to proceed with random simulation as described in the preceding section.

Backpropagation We consider both the sum and max-backup rules here. While the latter is typically considered to be overly aggressive and wasteful (of reward information), it is quite suitable for our setting and allows for a form of focused local search: if a high-reward candidate has been observed in a node's subtree, a max-backup encourages more future visits to the same node. This may lead to slight modifications to the candidate that bring about improved rewards.

4.3 Algorithm Engineering

Our implementation of BaMCTS exploits certain properties of the backdoor search problem to speed it up. One such property is that the root node of the branch-and-bound tree of each candidate evaluation is the same. Because solving the LP relaxation of the root node is typically much more time-consuming than other subproblems', we instrument CPLEX to solve the root LP once for all in advance and reuse its solution in subsequent evaluations. Rather than simulate and evaluate a single candidate at a time, we leverage the independence between the simulations to execute them and the candidate evaluation, in parallel.

5 Experiments

To evaluate our method, we designed a set of experiments to answer the following questions: **1) Backdoor Extraction:** Can BaMCTS find backdoors with better tree weight values (or rewards) compared to the biased random sampling of Dilkina et al. (2009)? **2) Sensitivity Analysis:** How sensitive is BaMCTS to its hyperparameters? **3) Suitability of Tree Weight as a Reward Function:** Is branching on higher tree weight backdoors conducive to smaller search trees or (for instances that are not solved to optimality within a time limit) a smaller optimality gap? In other words, is tree weight a legitimate reward function for BaMCTS?

5.1 Experimental Setup

Instances We conducted our experiments on the MIPLIB2017 Benchmark set (Gleixner et al. 2021) that contains 240 instances. We only considered the 164 mixed-binary instances, i.e., instances with no general integer variables, due to the ease of implementation of the tree weight for binary problems; extension to general integer variables is possible. We presolved the instances in advance to eliminate redundant variables and constraints, and also let CPLEX

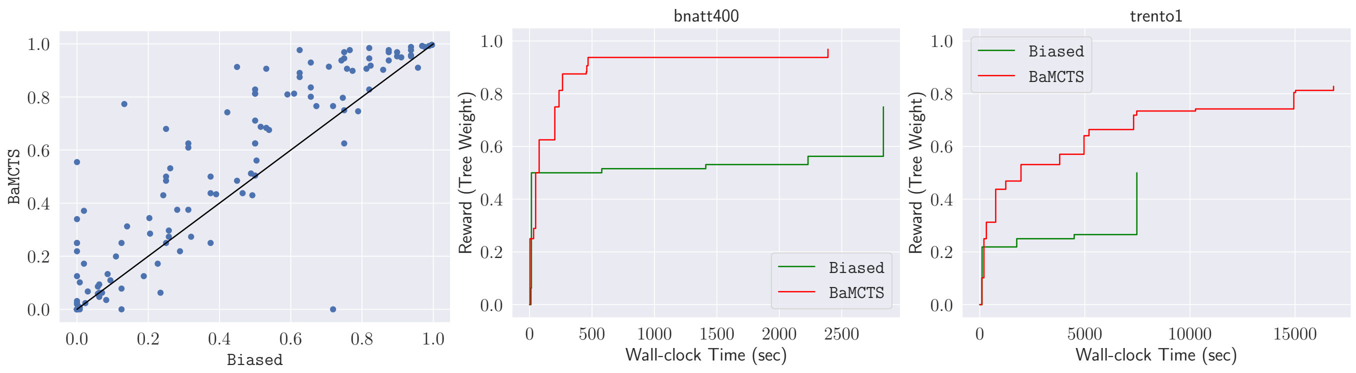

Figure 1: (left) BaMCTS outperforms Biased in finding backdoors with higher (mean) reward: most instances fall above the diagonal. (middle) & (right) BaMCTS attains higher rewards faster than Biased on two representative instances.

generate cuts at the root node; the resulting preprocessed instance was used instead of the original in all subsequent procedure, as presolving and root cuts are both standard steps in MIP solving, indepedently of any backdoor search. Some instances were excluded due to either memory issues during presolving or being solved at the root node without branching, reducing the final dataset size down to 142 instances.

Baselines We compared BaMCTS against two baselines: Biased and our implementation of SetCover from Fischetti and Monaci (2011). Each run consisted of two phases: backdoor search and backdoor-guided MIP solving.

Protocol for Backdoor Search For the search phase, each method was given a budget of 5 hours per instance. We recorded the sequence of improving backdoors found by each method on every instance, particularly the backdoor with the highest tree weight. For a fair comparison, we tasked all methods to find backdoors of a specific size K. Naturally, it is trivial to find large backdoors (at the extreme, the full integer set is a backdoor) but their usefulness in the downstream MIP solving phase diminishes as they get larger. We report our results for backdoor size of $K = 8$; however, we experimented with other backdoor sizes in the 5-10 range and the results still carry.

Protocol for MIP Solving In the second phase, we solved the same instance for one hour using CPLEX, but instrumented the solver to prioritize branching on the backdoor variables in the order they are given; this was achieved using CPLEX's branching priority feature. To mitigate CPLEX's randomness w.r.t. arbitrary initial conditions (Lodi and Tramontani 2013), we solve each instance with 3 random seeds.

BaMCTS Hyperparameters To test the sensitivity of BaMCTS, we sampled 30 hyperparameter configurations out of 128 configurations implied by the grid over:

- Backup $\in \{\text{sum}, \text{max}\}$;
- Expansion Type $\in \{\text{Best Score}, \text{Uniform}\}$;
- use_variance $\in \{\text{True}, \text{False}\}$;
- $\alpha_{\text{PC}} \in \{0.00, 0.01, 0.10, 0.50\}$;
- Exploration Parameter $(C) \in \{\frac{1}{\sqrt{2}}, 1, \sqrt{2}, 2, \sqrt{3}\}$.

For each configuration, we ran BaMCTS for one hour. We then selected the configuration with the highest mean tree weight. That single configuration (bottom row, Table 1) was then used for the 5-hour backdoor search introduced earlier.

Hardware All experiments were conducted on a large CPU cluster. All MIP solving runs use a single core with an 8GB memory limit and a 1-hour time limit. Backdoor search runs for both BaMCTS and Biased use 10 cores in parallel with a variable memory limit, capped at 63GB, that is proportional to the instance's size on disk, and a 5-hour time limit for the selected hyperparameter configuration.

5.2 Results

Backdoor Extraction Performance Figure 1 compares the performance of BaMCTS vs. Biased in terms of the quality of the backdoors they find within the 5-hour time limit per instance. BaMCTS clearly outperformed Biased, leading to discovery of higher tree weight backdoors on many more instances. This is further demonstrated in the top-right plot of Figure 2. There, we observe that BaMCTS with even 1 hour time budget (blue) generally outperformed Biased with 5 hour budget (green). BaMCTS with 5 hour budget (red) further widened the gap. Note that SetCover does not depend on tree weight as a reward and is thus not comparable with BaMCTS here.

Sensitivity Analysis The box-plots of Figure 2 show the per-instance sensitivity of BaMCTS to the set of 30 hyperparameter configurations we tested. The instances are sorted by median reward from low (hard instances) to high (easy instances). Even though we observe quite high sensitivity for some instances, for the majority of them most configurations obtain similar reward. Figure 3 shows all 30 hyperparameter configurations for BaMCTS as well as the average reward that each configuration achieved over the entire dataset. Most of the configurations resulted in average reward in the range of [0.28, 0.35], indicating that at least on average (across all instances) there is not a substantial variability in terms of the reward, and BaMCTS can be used out of the box with default hyperparameter values and achieve a reasonable performance. Note that MIPLIB2017 is quite

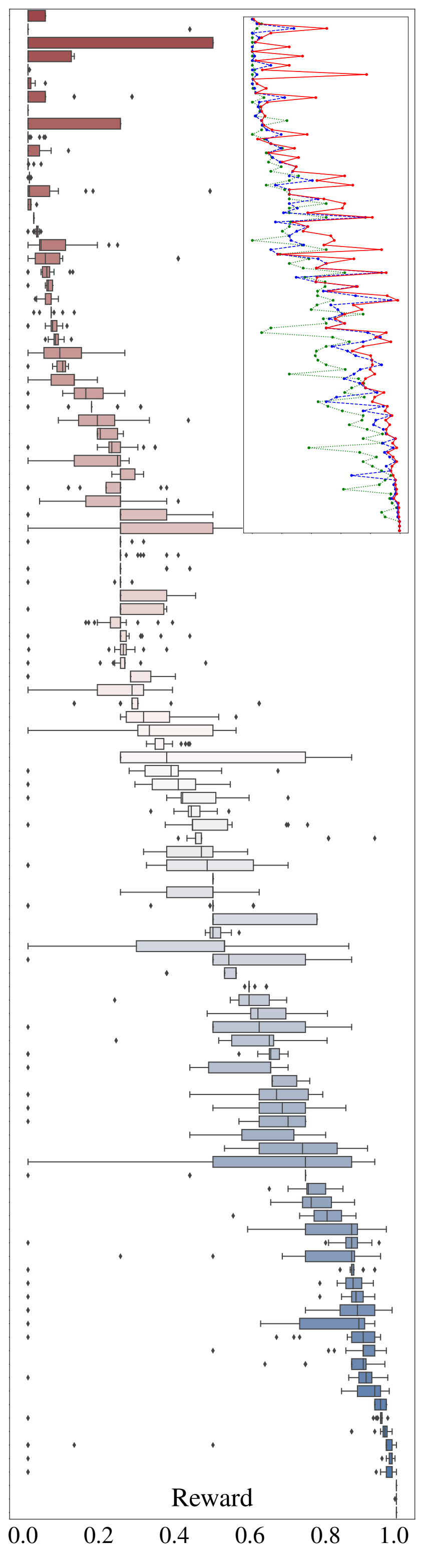

Figure 2: Box-plots showing the distribution of rewards for each instance over the 30 hyperparameter configs. Instances are sorted based on median reward, hard (red) to easy (blue). *top-right*: Reward comparison between `Biased` (green) and `BaMCTS` run with 1 (blue) & 5 (red) hours.

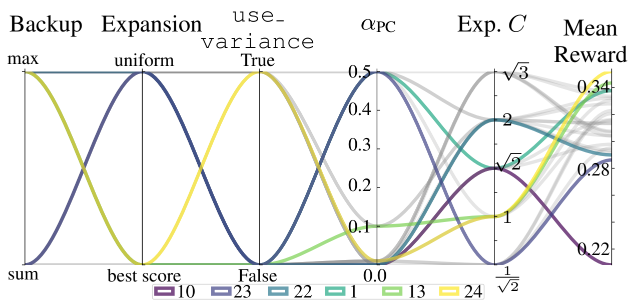

Figure 3: `BaMCTS` is robust w.r.t. hyperparameter changes. Each curve in the figure represents a hyperparameter configuration (30 total) and indicates the values taken by that config as well as the obtained mean reward after running `BaMCTS` with that config on the dataset for 1 hour. Notice the concentration of rewards within the [0.28, 0.35] band. This shows that for most configs the reward does not change drastically. The highlighted curves represent the best and worst performing configs (see Table 1 bellow).

	Backup	Exp. Type	`use_variance`	α_{PC}	C	Mean Reward
10	max	Uniform	False	0.00	$\sqrt{2}$	0.208
23	sum	Uniform	False	0.50	$1/\sqrt{2}$	0.286
22	max	Uniform	False	0.00	2	0.289
20	sum	Uniform	True	0.10	2	0.291
11	max	Uniform	True	0.00	2	0.293
19	max	Uniform	False	0.01	$1/\sqrt{2}$	0.295
21	sum	Uniform	False	0.00	$\sqrt{2}$	0.297
16	sum	Uniform	False	0.50	$\sqrt{3}$	0.304
12	max	Uniform	True	0.50	2	0.304
7	max	Uniform	True	0.00	$\sqrt{3}$	0.306
3	max	Uniform	False	0.01	$\sqrt{3}$	0.307
9	max	Uniform	True	0.10	2	0.313
0	sum	Uniform	False	0.01	2	0.313
17	sum	Uniform	False	0.50	1	0.315
6	sum	Uniform	True	0.00	$\sqrt{2}$	0.315
26	max	Uniform	True	0.01	$1/\sqrt{2}$	0.320
2	max	Uniform	True	0.00	$1/\sqrt{2}$	0.322
4	sum	Best	False	0.00	2	0.327
25	max	Best	False	0.01	1	0.330
15	max	Best	True	0.50	2	0.332
5	max	Best	False	0.01	$\sqrt{3}$	0.333
8	sum	Best	False	0.01	$\sqrt{3}$	0.337
1	max	Best	False	0.50	$\sqrt{2}$	0.337
13	max	Best	False	0.10	1	0.343
24	max	Best	True	0.01	1	0.351

Table 1: All `BaMCTS` parameter configurations sorted by mean reward (tree weight); higher is better.

diverse and it is likely that testing on a more homogeneous dataset would result in even lower parameter sensitivity.

Tree Weight and Branch-and-Bound Tree Size Table 2 summarizes the MIP solving results with a 1-hour time limit. CPX–`def` is CPLEX 12.10.0 with traditional branch-and-bound (i.e., with CPLEX's "dynamic search" turned off);

	seed 1 (47, 56, 4)		seed 2 (45, 61, 7)		seed 3 (46, 60, 6)	
	CPX–def	CPX–BaMCTS	CPX–def	CPX–BaMCTS	CPX–def	CPX–BaMCTS
# of nodes (solved by both)	6902.7	**6097.7**	6173.8	**5030.1**	7744.9	**7001.9**
total time (solved by both)	179.5	**175.9**	156.6	**138.3**	169.4	**156.8**
optimality gap (not solved by either)	23/56	**33/56**	24/61	**37/61**	20/60	**40/60**
# of instances (solved by one method)	**3**	1	**4**	3	3	3

Table 2: Summary of MIP solving results. "seed 1 (47, 56, 4)" means that for this seed, 47 instances were solved by both CPX–def and CPX–BaMCTS, 56 were not solved by either within 1 hour, and 4 were solved by only one of the two. Geometric means are reported for "# of nodes" and "total time"; the number of wins/losses are reported for "optimality gap". The better method for each metric is in bold.

results with "dynamic search" are similar. CPX–BaMCTS is CPLEX 12.10.0 with branching priorities defined by the best backdoor found for that instance by the best hyperparameter configuration of BaMCTS (configuration **24** in Table 1). Of the initial 142 instances, we restrict the MIP solving here to 115 instances for which BaMCTS found at least one backdoor candidate with non-zero tree weight. Within each random seed, the instances are partitioned into three sets: (i) instances solved by both methods: here we compare the shifted geometric means of the number of nodes and total time (lower is better; see page 33 of (Hendel 2015) for further discussion of the suitability of this metric), with shifts 100 and 10, respectively; (ii) instances that are not solved by either method: here the optimality gap after the 1-hour limit tells us which method made more progress; we count the number of instances for which each method achieved a smaller gap (higher is better); and (iii) instances that are solved by only one of the methods: here we simply count the number of wins for each method (higher is better).

Table 2 shows that CPX–BaMCTS wins on the first three metrics consistently across the three seeds, with an average reduction of 700 to 1100 in the number of nodes. The last metric (final row) records two wins for CPX–def and one tie, but it is over 4 to 7 instances out of more than 110. We have also used the two-sided Wilcoxon signed-rank test, as described in (Hendel 2014), to compare the distribution of values for the three metrics, and have found the corresponding p-values to be typically small, implying that the observed values for these metrics come from distributions with different medians for each of CPX–def and CPX–BaMCTS; this agrees with the conclusions from Table 2.

The fact that branching with the best backdoor found by BaMCTS typically leads to better MIP solving (smaller number of nodes, smaller optimality gaps, etc.) confirms that tree weight is an informative reward signal for backdoor search.

Lastly, we note that we have attempted to compare against the SetCover method of Fischetti and Monaci (2011) on the MIP solving metrics. However, SetCover fails to return a size-8 backdoor for half of the 142 instances, requiring more variables to cover fractional vertices. This makes it difficult to compare BaMCTS to SetCover; modifications to the latter are necessary but are out of scope for this paper.

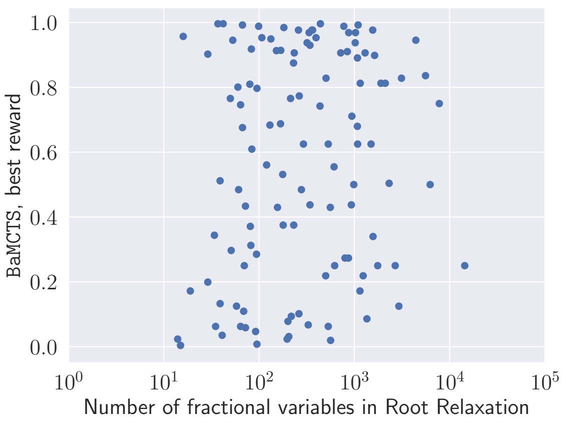

Figure 4: Scatter plot of the number of fractional variables in the MIP root LP relaxation solution vs. the best reward (i.e., tree weight) found by our method.

Impact on Total Time While Table 2 shows that branching with the backdoors found by our method result in only small reductions in total time, we note that: (1) this reduction is consistent across the three seeds and is in the 2-10% range, which is non-trivial for standard MIPLIB2017 Benchmark instances; and (2) the experiment aims primarily at showing that branching with the backdoors translates into a smaller *number of nodes*, a metric for which improvements are larger. Because CPLEX is not open-source, we are unable to control its behavior beyond its callbacks, but we believe that further total time speedups may be achieved with tighter integration with the internal solver code.

Tree Weight vs. Number of Variables Given the wide range of tree weight values that were observed as a result of backdoor search, we were interested in whether only instances with small action spaces, i.e., a small number of integer variables that are fractional in the solution of the root LP relaxation of branch-and-bound tree, were associated with tree weights (rewards) that are close to 1. Figure 4 shows that is not the case at all: the tree weight values are spread across the wide range of values on the horizontal axis, hence our results are not biased towards "easy" small instances.

6 Conclusion

In this paper we proposed BaMCTS, a Monte Carlo Tree Search framework for finding strong backdoors in MIPs and demonstrated through our experiments its merits relative to earlier backdoor search algorithms in MIP literature. We hope that this paper would motivate further research in this direction and position MCTS as a viable approach in finding MIP backdoors. One particularly interesting future direction is to apply this technique on more homogeneous families of instances instead of MIPLIB to discover inherent structures for those problem families by studying their backdoors. We are planning to release our code to streamline these efforts.

Acknowledgements

Elias B. Khalil acknowledges support from the Scale AI Research Chair Program and an NSERC Discovery Grant. Bistra Dilkina acknowledges support from NSF AI Institute for Advances in Optimization Award #2112533.

References

Abe, K.; Xu, Z.; Sato, I.; and Sugiyama, M. 2019. Solving NP-Hard Problems on Graphs with Extended AlphaGo Zero. *arXiv preprint arXiv:1905.11623*.

Achterberg, T.; and Berthold, T. 2009. Hybrid Branching. In *CPAIOR*, 309–311. Springer.

Achterberg, T.; Koch, T.; and Martin, A. 2005. Branching Rules Revisited. *Operations Research Letters*, 33(1): 42–54.

Achterberg, T.; Koch, T.; and Martin, A. 2006. MIPLIB 2003. *Operations Research Letters*, 34(4): 361–372.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-Time Analysis of the Multiarmed Bandit Problem. *Machine learning*, 47(2): 235–256.

Berthold, T. 2014. Rens. *Mathematical Programming Computation*, 6(1): 33–54.

Bertsimas, D.; Griffith, J. D.; Gupta, V.; Kochenderfer, M. J.; Mišić, V. V.; and Moss, R. 2014. A Comparison of Monte Carlo Tree Search and Mathematical Optimization for Large Scale Dynamic Resource Allocation. *arXiv preprint arXiv:1405.5498*.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1): 1–43.

Cook, W. 2012. Markowitz and Manne + Eastman + Land and Doig = Branch and Bound. *Optimization Stories*, 227–238.

Couëtoux, A.; Hoock, J.-B.; Sokolovska, N.; Teytaud, O.; and Bonnard, N. 2011. Continuous Upper Confidence Trees. In *International Conference on Learning and Intelligent Optimization*, 433–445. Springer.

Coulom, R. 2007. Computing "Elo Ratings" of Move Patterns in the Game of Go. *ICGA journal*, 30(4): 198–208.

Dilkina, B.; Gomes, C. P.; Malitsky, Y.; Sabharwal, A.; and Sellmann, M. 2009. Backdoors to Combinatorial Optimization: Feasibility and Optimality. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 56–70. Springer.

Fischetti, M. 2014. BRANCHstorming (Brainstorming About Tree Search). http://www.dei.unipd.it/~fisch/papers/slides/2014%20ISCO%20%5bplenary%20Fischetti%20BRANCHstorming%5d.pdf. Accessed: 2022-04-13.

Fischetti, M.; and Monaci, M. 2011. Backdoor Branching. In *International Conference on Integer Programming and Combinatorial Optimization*, 183–191. Springer.

Fischetti, M.; and Monaci, M. 2014. Exploiting Erraticism in Search. *Operations Research*, 62(1): 114–122.

Gaudel, R.; and Sebag, M. 2010. Feature Selection as a One-Player Game. In *International Conference on Machine Learning*, 359–366.

Gelly, S.; and Silver, D. 2007. Combining Online and Offline Knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, 273–280.

Gleixner, A.; Hendel, G.; Gamrath, G.; Achterberg, T.; Bastubbe, M.; Berthold, T.; Christophel, P.; Jarck, K.; Koch, T.; Linderoth, J.; et al. 2021. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 1–48.

Hendel, G. 2014. *Empirical Analysis of Solving Phases in Mixed Integer Programming*. Master's thesis, Technische Universität Berlin.

Hendel, G. 2015. Enhancing MIP Branching Decisions by Using the Sample Variance of Pseudo Costs. In *Integration of AI and OR Techniques in Constraint Programming*, volume 9075, 199 – 214. In press.

Hendel, G.; Anderson, D.; Le Bodic, P.; and Pfetsch, M. E. 2021. Estimating the Size of Branch-and-Bound Trees. *INFORMS Journal on Computing*.

Kilby, P.; Slaney, J.; Thiébaux, S.; Walsh, T.; et al. 2006. Estimating Search Tree Size. In *Proc. of the 21st National Conf. of Artificial Intelligence, AAAI, Menlo Park*.

Kocsis, L.; and Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. In *ECML*, 282–293. Springer.

Kottler, S.; Kaufmann, M.; and Sinz, C. 2008. Computation of Renameable Horn Backdoors. In *International Conference on Theory and Applications of Satisfiability Testing*, 154–160. Springer.

Li, Z.; and Van Beek, P. 2011. Finding Small Backdoors in SAT Instances. In *Canadian Conference on Artificial Intelligence*, 269–280. Springer.

Linderoth, J. T.; and Savelsbergh, M. W. 1999. A Computational Study of Search Strategies for Mixed Integer Programming. *INFORMS Journal on Computing*, 11(2): 173–187.

Lodi, A.; and Tramontani, A. 2013. Performance Variability in Mixed-Integer Programming. *Tutorials in Operations Research: Theory Driven by Influential Applications*, 1–12.

Loth, M.; Sebag, M.; Hamadi, Y.; and Schoenauer, M. 2013. Bandit-based Search for Constraint Programming. In *International Conference on Principles and Practice of Constraint Programming*, 464–480. Springer.

Nemhauser, G. L.; and Wolsey, L. A. 1988. *Integer and Combinatorial Optimization*. John Wiley & Sons.

Paris, L.; Ostrowski, R.; Siegel, P.; and Sais, L. 2006. Computing Horn Strong Backdoor Sets Thanks to Local Search. In *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, 139–143. IEEE.

Rimmel, A.; Teytaud, F.; and Cazenave, T. 2011. Optimization of the Nested Monte-Carlo Algorithm on the Traveling Salesman Problem with Time Windows. In *European Conference on the Applications of Evolutionary Computation*, 501–510. Springer.

Sabharwal, A.; Samulowitz, H.; and Reddy, C. 2012. Guiding Combinatorial Optimization with UCT. In *International conference on integration of artificial intelligence (AI) and operations research (OR) techniques in constraint programming*, 356–361. Springer.

Satomi, B.; Joe, Y.; Iwasaki, A.; and Yokoo, M. 2011. Real-Time Solving of Quantified CSPs based on Monte-Carlo Game Tree Search. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the Game of Go Without Human Knowledge. *Nature*, 550(7676): 354–359.

Szeider, S. 2005. Backdoor Sets for DLL Subsolvers. *Journal of Automated Reasoning*, 35(1-3): 73–88.

Williams, R.; Gomes, C. P.; and Selman, B. 2003. Backdoors to Typical Case Complexity. In *IJCAI*, volume 3, 1173–1178.

Xu, R.; Kadam, P.; and Lieberherr, K. 2021. First-Order Problem Solving through Neural MCTS based Reinforcement Learning. *arXiv preprint arXiv:2101.04167*.